Training Distributed Garbage: The DMOS Collector

Richard L. Hudson,[¥] Ron Morrison,[†] J. Eliot B. Moss,[¥] & David S. Munro[†]

[¥]Department of Computer Science, University of Massachusetts,
Amherst, MA 01003, U.S.A.
Email: {hudson, moss}@cs.umass.edu

[†]School of Mathematical and Computational Sciences,
University of St Andrews, North Haugh, St Andrews, Fife, KY16 9SS, Scotland
Email: {ron, dave}@dcs.st-and.ac.uk

Abstract

A new garbage collection algorithm for distributed object systems, called DMOS (Distributed Mature Object Space), is presented. It is derived from two previous algorithms, MOS (Mature Object Space), sometimes called the train algorithm, and PMOS (Persistent Mature Object Space). The contribution of DMOS is that it provides the following unique combination of properties for a distributed collector: safety, completeness, non-disruptiveness, incrementality, and scalability. Furthermore, the DMOS collector is non-blocking and does not use global tracing. A complete description of the DMOS collector is given, together with its associated correctness arguments and its relationship to other work.

1 Introduction

Automatic storage management is an essential property of high level programming systems providing an error-free abstraction with which the programmer may manipulate space without regard to the inessential details of physical storage. The abstraction holds only until the store becomes full. Thus it is important for the storage management system to distinguish useful data from garbage, so that the space occupied by the garbage may be reused. The technique used to identify the unreferenced space automatically is called *garbage collection* (see Wilson [Wilson92] for a survey of these techniques).

Here we are concerned with garbage collection in a distributed system where each node in the system has its own local storage and may communicate with other nodes only by passing messages. The problem is difficult because of asynchrony, implying lack of knowledge of global state, and lack of globally atomic operators on that state.

We present a new garbage collection algorithm for distributed object systems, called DMOS (Distributed Mature Object Space). It derives from both the MOS (Mature Object Space) [HM92], sometimes known as the train algorithm, and PMOS (Persistent Mature Object Space) [MMH96] collectors. The MOS collector is an incremental main memory copying collector specifically designed to collect large, older generations of a generational scheme in a non-disruptive manner. The PMOS collector extends MOS to ensure incrementality in a persistent context, while also limiting the I/O overhead.

The contribution of the DMOS collector is the following unique combination of properties for a distributed collector without the need for global tracing:

- **Safety**: the collector does not collect live (reachable) objects.
- **Completeness**: the collector is complete in that all garbage, including cyclic garbage that spans nodes, is collected within a finite number of invocations.
- **Non-disruptiveness**: the collector bounds the amount of collection work, thereby bounding the time and space requirements, for each invocation.
- **Incrementality**: the collector reclaims space incrementally without global knowledge of reachability.

- **Scalability**: the collector is potentially scalable in that it is decentralised, uses asynchronous communication, and has no protocols that demand the involvement of all nodes.
- **Non-blocking**: the collector at a node need only synchronise with other nodes in a few cases. Application computation never need wait for such collector synchronisation.

The collector assumes the following support in delivering the above:

- Each node in the system has its own local storage and may communicate with other nodes only by passing messages.
- Ordered delivery of messages is guaranteed without omission, corruption, or duplication. Causal delivery is not assumed.
- Nodes appear to operate correctly, without crashes or Byzantine behaviour.
- No bounds are placed on the relative rates of computation of the nodes.
- Events and actions at a given node are totally ordered, yielding a partial ordering of events in the system as a whole.

As presented, the collector is well suited to distributed memory multiprocessors and to applications that do not require fault-tolerance. To widen the applicability of the collector, fault-tolerance may be provided by lower levels of the system. While others have chosen to build some of this support into garbage collectors, we regard these facilities as being provided by lower level protocols, upon which the garbage collector can be built, in order to separate policy and mechanism and to keep the levels of abstraction relatively understandable.

1.1 The Computational Model

Our distributed computational model is made up of computation, objects, and pointers.

Computation consists of one logical *process*, perhaps with concurrent threads, per (logical) node with the processes communicating via *messages*. Computation proceeds by creating and mutating *objects*. A physical node may support more than one logical node.

A (physical) object resides on a single node, which we term the object's *home*, and may contain any number of pointers (references to other objects) as well as non-pointer data. DMOS moves (logical) objects within nodes and allows applications to move objects across nodes, so it includes algorithms to *substitute* one physical object for another and to update the affected references.

Each node has zero or more *root pointers* to objects, which we can view as being part of the node's process's state. If an object is not reachable via a chain of pointers through objects originating from some root then the object is garbage and may be reclaimed. Pointers may propagate to other nodes, and be stored there, by being included in messages.

1.2 Overview

Our goal is a fully distributed algorithm that will, concurrently and incrementally, eventually detect and reclaim all garbage while computation continues. Since DMOS builds on MOS and PMOS, we describe it first by giving a concise summary of the MOS collector, indicating how DMOS differs at a high level but omitting details of distribution. The high level description is followed by presentations of several detailed protocols, for keeping track of pointers to objects (and detecting when objects are unreachable), for adjusting pointers when objects are moved within (or across) nodes, and for managing the internal data structures (trains and cars) of the DMOS collector. The presentation includes arguments for correctness of the protocols, and safety and completeness of the collector as a whole.

Since object migration is a policy decision in a distributed computation that mutators may not wish imposed upon them, we will not consider it as a fundamental technique required by the garbage collector. However, our collector does allow objects to migrate.

2 The DMOS Collector

The DMOS collector is described, in the manner similar to the MOS and PMOS descriptions, using the metaphor of *trains* made up of *cars*. The address space of each node is divided into a number of disjoint blocks (cars). One car is collected in each local invocation of the collector, by copying its potentially reachable objects into other cars. Since only potentially reachable data is copied, all garbage contained within one car will be collected immediately. The number of cars per node and the individual size of the cars is a matter of implementation policy, but for each invocation of the collector the car size bounds the time and space required for that invocation.

To collect cyclic garbage that spans more than one car, cars are grouped together into trains. Each car resides on a single node but a train may span more than one node. It is again a matter of policy as to how many trains there are (there must be at least two) and when new trains are created. By ensuring that all the cars in a train are collected by copying the potentially reachable data into other trains, cyclic garbage will be left behind and can be collected, if it can be marshalled into the same train. The trick is to find the minimum constraint on the order of collection to guarantee completeness. For this it is sufficient to order the trains in terms of the (logical) time they are created. Hence we will refer to trains being *older* or *younger* than other trains.

The DMOS collector uses the following rules for copying data from a car during collection:

1 Data locally reachable[1] from roots is copied to a younger train, adding a car to that train if required.

2 Data locally reachable from younger trains is copied to those trains, adding a car if required. If an object is reachable from more than one younger train, it may be copied to any younger train from which it is reachable.

3 Data locally reachable from older trains is copied to any other car of its current train, adding a car if required.

4 Data locally reachable from other cars of the same train is copied to any other car of the train, adding a car if required.

5 The remaining data is unreachable and is reclaimed immediately.

It should be noted that the above rules are followed in order. To complete the collection of cyclic garbage one more rule is required:

0 If no object in a train is reachable from outside the train, reclaim the entire train. If necessary, create another train to ensure that there are always at least two trains.

The algorithm allows any car from any train to be selected for collection.[2]

Figure 1 illustrates the algorithm, showing a sequence of four collections, which collects intra-train and inter-train cycles of garbage, and reclusters the live objects.

Since in DMOS trains may span nodes,[3] in following Rule 2 the collector may find that it wishes to add a car to a train which is not represented on this node. Thus Rule 2 must be amended as follows:

---

[1] Object Y is locally reachable from pointer X if X refers to Y, or there is a chain of pointers, all within the car, that leads from X to Y.

[2] The work of Cook, Wolf, and Zorn [CWZ94] suggests that a flexible selection policy allowing a collector to choose which partition to collect can significantly increase the amount of space reclaimed and, in an OODB context, reduce the amount of I/O.

[3] An alternative is to restrict trains to a single node. This, however, has the consequence to making object migration compulsory if distributed cyclic garbage is to be marshalled into a single train.

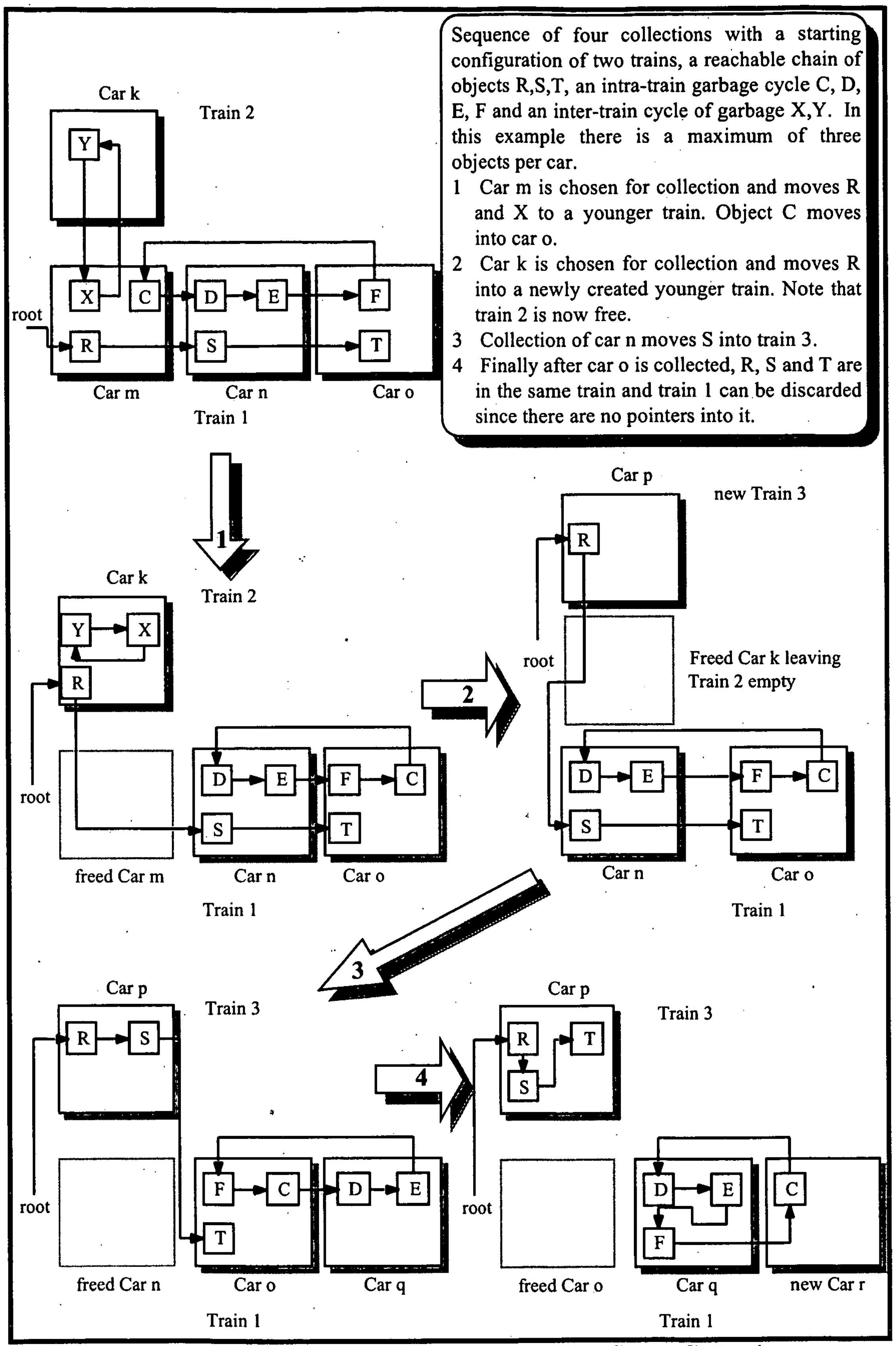
2 Data locally reachable from younger trains is copied to those trains, adding a car if required. If an object is reachable from more than one younger train, it may be copied to any younger train from which it is reachable. If the destination train is not represented on this node, then the node should *join* the train and create a new car in that train.

This, as we will see later, requires a distributed protocol as does the detection of an empty train, Rule 0.

The completeness of the algorithm is based on four constraints:

- Objects never move from a younger train to an older one,
- garbage can never move to a train younger than its youngest referent (as of some time), and
- each car is eventually collected, which implies that
- the oldest train is eventually evacuated.

A completeness argument for the distributed collector, which also addresses the anomaly discovered by Seligmann and Grarup [SG95], is given later.



Sequence of four collections with a starting configuration of two trains, a reachable chain of objects R,S,T, an intra-train garbage cycle C, D, E, F and an inter-train cycle of garbage X,Y. In this example there is a maximum of three objects per car.

1 Car m is chosen for collection and moves R and X to a younger train. Object C moves into car o.
2 Car k is chosen for collection and moves R into a newly created younger train. Note that train 2 is now free.
3 Collection of car n moves S into train 3.
4 Finally after car o is collected, R, S and T are in the same train and train 1 can be discarded since there are no pointers into it.

Figure 1: Example Sequence of Mature Object Space Collection

3 Addressing Objects

Since DMOS is a copying collector, it moves objects, and is thus involved in issues of object addresses and locations. Put another way, in DMOS objects are referred to with addresses that encode at least a logical location, because location (e.g., within a car or train) is fundamental to how the collector works. The MOS collector assumes that each object reference somehow encodes the car and train containing the referent object. This might be accomplished using tables that map regions of address space to cars and trains. DMOS assumes that references also somehow encode the home node of the referent object.

Object references might include an absolute address, or might be based more on a location independent object identifier. However, when an object is moved, references to its old copy may survive for some time, so one must either defer reusing the address space containing the old copy (not an attractive alternative), or arrange that a complete object reference includes information beyond a node and absolute address at that node. We assume that car numbers at a node are not reused, or else reused quite slowly. The car number can then distinguish different periods of use for the same portion of absolute address space allowing prompt reuse of vacated memory.

4 Pointer Tracking

Collecting a car requires knowing external references (from objects outside the car to objects inside the car). To avoid global synchronisation we cannot demand that such knowledge be entirely up-to-date. On the other hand, safety requires that the collector never treats a reachable object as unreachable, and completeness requires that a node eventually discovers when there are no longer external references to a local object.

The pointer tracking algorithm is designed to meet the above criteria by ensuring that the home node H, of an object o, is informed of any relevant manipulation of a pointer to o by another node. This ensures that H has sufficient information to allow for either object substitution or reclamation of the object. As we will see, object substitution requires H to know which other nodes refer to o, and for reclamation H must know that there are no pointers to o from other nodes.[4]

We defer discussion of the correctness of the pointer tracking until after its description.

4.1 Events Related to Pointer Tracking

Our pointer tracking mechanism consists of handling five events. The home node, H, is informed of these events via asynchronous messages. Our initial description of these events will be presented with a virtual message being generated for each event in the system. This will be followed by an optimisation strategy describing how the number of messages and the size of each message may be reduced. The five events are detailed in Table 1.

[4] The protocol is designed specifically to avoid any need for causal messaging [Fidge96].

Event	Description
<s, n, o, A, B>	This (send) event indicates that node A has sent to node B a pointer to o. The number n is chosen such that no other event <s, m, o, A, B> has m = n, i.e., n, o, A, and B together uniquely identify the *s* event for all time and space. This event is said to *happen at* A (the sender), hence it is A's responsibility to inform H. Intuition: This indicates that a new pointer to o has been created in the virtual channel A→B. The number n is used to match *s* events with their corresponding *r* events.
<r, n, o, A, B>	This (receive) event indicates that node B has received a pointer to o sent by node A. The number n is chosen to match the corresponding *s* event. This event happens at B. Intuition: An *r* event indicates that the pointer has been deleted from the virtual channel A→B and has been created in a message receive buffer at B.
<d, n, o, A, B>	This (delete) event indicates that node B has deleted from its message buffers the received pointer to o uniquely identified by n, A, and B. The number n corresponds to the *s* and *r* messages previously described. This event happens at B.
<+, m, o, A>	This event indicates that node A has created a new pointer, uniquely identified by m, to object o. This event happens at A.
<–, m, o, A>	This event indicates that node A has deleted the specific pointer, uniquely identified by m, to object o. This event happens at A.

Table 1: Pointer Tracking Events

Figure 2 illustrates a possible scenario of events in which a pointer is sent from one node to another. Messages sent are drawn as broken arrows and object pointers signified by unbroken arrows. Node A sends a copy of the pointer to node B and eventually informs H of the *s* event. Node B receives the pointer in its receive buffer, and informs H of the *r* event. Node B puts the pointer into its heap,[5] eventually informing H of the + event. Finally, B deletes the pointer from its receive buffer and eventually informs H of the *d* event.

There is no specific requirement for rapid delivery to H of information about events at A and B. This is intentional, since such information can normally be piggy-backed on other communication, thereby reducing the overhead of the scheme. Messages 2, 3, and 4 in Figure 2 arrive at H in that order. There is, however, no constraint on the arrival of message 1, relative to messages 2, 3, and 4.

[5] Note that this is optional, and hence the * in the diagram.

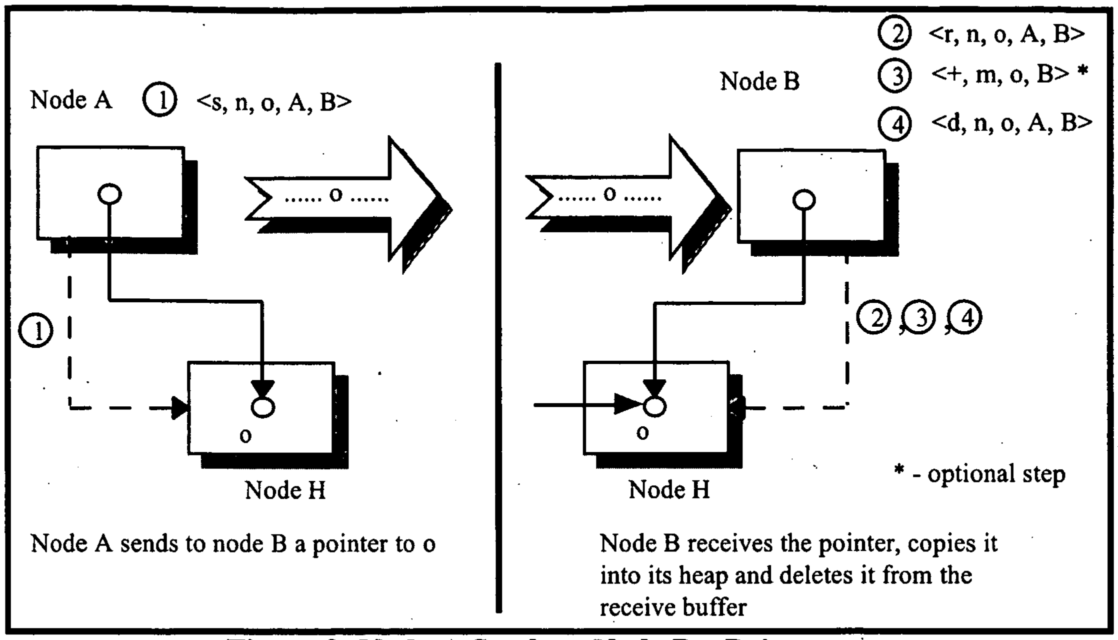


Figure 2: Node A Sends to Node B a Pointer to o

In Table 1, either A or B can be the home node, H. We assume, however, that H is informed immediately of events that happen at H.

It would appear at first glance that the number of events may be reduced by, for example, combining *d* and – or *r* and +. For the moment we find it convenient to maintain the distinction but will return to this point later.

4.2 Constraints on Ordering of Events

To describe the correct operation of the system there are a number of constraints on the order in which the five kinds of events can occur. For this we introduce a precise definition of the predicate any(o, Y, E), which indicates whether node Y has any pointers to object o in the situation described by the set of events E.

Definition: Given a set of events E, where each event is of the form described in Table 1, any(o, Y, E) for an object o and node Y holds iff E contains an event <r, n, o, X, Y> but not the event <d, n, o, X, Y>, or E contains an event <+, n, o, Y> but not the event <–, n, o, Y>.

The intuition is that a node has a pointer if it has received it in a message buffer but not yet deleted it, or created it in the heap but not yet destroyed it.

The legal event sets for a system are defined recursively, in terms of the events that may legally be added to a given event set E, as defined in Table 2:

Rule	Intuition
The empty event set is legal.	Initial state.
H adds to E a <+, m, o, A> event not in E when object o is created at H.	The home node can create objects.
If any(o, A, E) is true then A can add an <s, n, o, A, B> event not in E.	If A has a pointer to o then it can send it to any other node.
If event <s, n, o, A, B> is in E but event <r, n, o, A, B> is not, then the *r* event may be added to E.	If a pointer has been sent but not yet received then the *r* event can occur.
If event <r, n, o, A, B> is in E but event <d, n, o, A, B> is not, then the *d* event may be added to E.	If a pointer has been received but not deleted from the message buffer then it may be so deleted. Note that before the *d* event any(o, B, E) is true.
If any(o, A, E) then a <+, m, o, A> event not in E may be added to E.	If A has a pointer to o then it may create a heap copy of the pointer.
If <+, m, o, A> is in E but <–, m, o, A> is not, then the – event may be added to E.	If A has a heap copy of a pointer that it has not yet deleted then it may delete it. Note that before the – event any(o, A, E) is true.

Table 2: Legal Event Sets

4.3 Correctness

The correctness criterion for collection depends upon being able to determine that no other node has pointers to a given object. To establish this we require a further definition.

Definition: *absence(o, E)* is true iff any(o, X, E) is false for all nodes X other than H, the home node of o, and E has no event <s, n, o, A, B> such that the corresponding receive event <r, n, o, A, B> is not in E.

Claim: Let E be the set of events known at H involving o; we call this H's view. If absence(o, E) then no node other than H has a pointer to o.

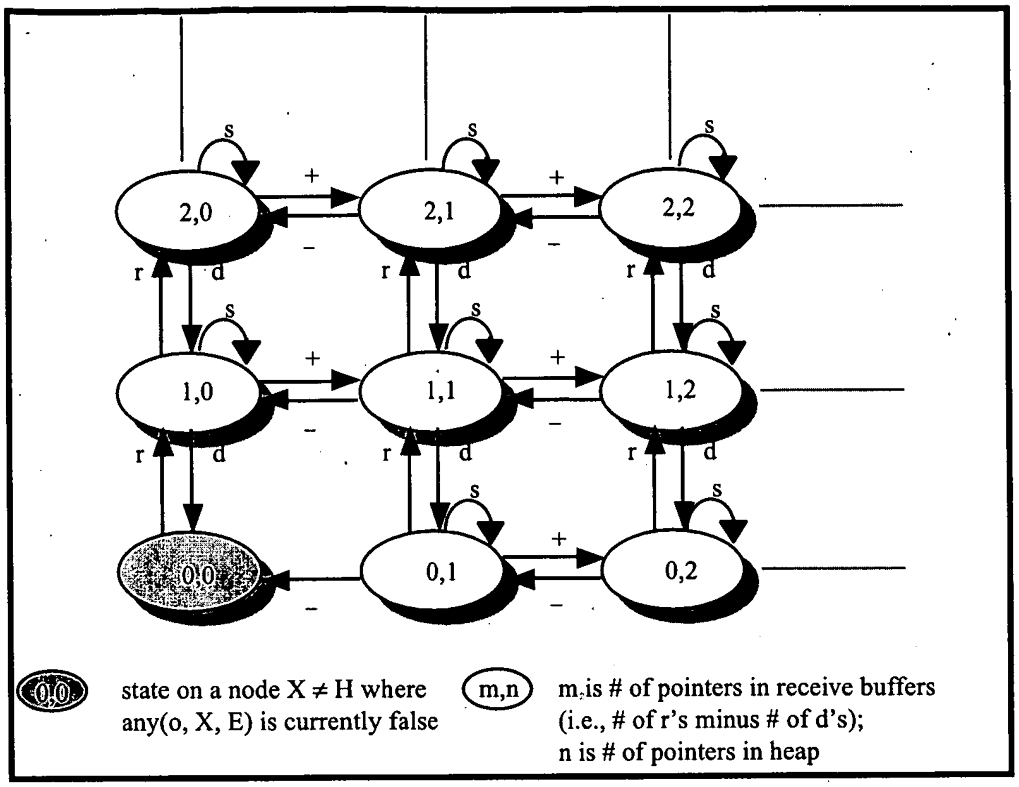

2,0
2,1
2,2
1,0
1,1
1,2
0,0
0,1
0,2
s
+
–
r
d
0,0
state on a node X ≠ H where any(o, X, E) is currently false
m,n
m is # of pointers in receive buffers (i.e., # of r's minus # of d's); n is # of pointers in heap

Figure 3: Node Event State Diagram for Each Object

Proof: The intuition is that if in H's view no other node has a pointer and there are no pointers in transit, then H's view is correct and is up to date.

Before tackling the proof itself, it is helpful to examine the state diagram presented in Figure 3, which shows the possible states of a node X ≠ H with respect to the number of copies it may have of a given pointer, in X's receive buffers (*m*) and in X's heap (*n*). The *s*, *r*, *d*, +, and – arrows signify send, receive, delete, and +/– events occurring at X. Note that in states where *m* is 0, delete events are never generated since there are no pointers in the receive buffers. Similarly when *n* is 0, no – (minus) events occur since there are no pointers in the heap. In the initial (and final) state ($m = 0$, $n = 0$) the only event that can occur on this node is a receive event.

Firstly, observe that the legal event rules constrain adding events that happen at a given node X primarily in terms of what has already happened at X. The sole exception is *r* events, which require a matching *s* event, which generally occurs at a different node.

Secondly, observe that since message delivery is ordered, H's view of what has happened at a node X is a prefix of what has actually happened at X.

Consider any node X other than H. Now assume that absence(o, E) is true, implying that in H's view, X's state machine is in state (0, 0). Thus any(o, X, E) is false. Further, any(o, X, E|X) is also false, where E|X means the set of events in E that happened at X.[6] E|X is a prefix of what has actually happened at X, but since any(o, X, E|X) is false, the only legal event that could be added to E|X is an *r* event. Thus the only legal next event at any node in H's view is an *r* event.

Since all *s* events in H's view are matched with corresponding *r* events, the only possible source of a new *s* event is H. We observe that H's view of itself is necessarily up to date, so H can be the only node possessing a pointer to o. Put another way, since no other node can legally add an *s* event, no other node possesses a pointer to o.

[6] This is easily seen from the definition of any(o, X, E) since it refers only to events that happen at X.

Our conclusion relies only on accepting the definition of any(o, X, E) as corresponding to our informal notion of X possessing any pointers to o in the situation described by E, the ordered and reliable delivery of messages between any pair of nodes, and the definition of legal event sets.

4.4 Optimisations to the Pointer Tracking Algorithm

We consider six optimisations to the pointer tracking algorithm: removing unique numbers in events; reducing the number of messages; reducing the bookkeeping at H; piggy-backing messages; compressing multiple event information into messages; and combining *d* and – events.

4.4.1 Removing the unique numbers from events

We argue that the unique numbers (*n* and *m* in Table 1) can be removed by counting the number of events of similar form to obtain an equivalent pointer tracking algorithm.

For *s* and *r* events related to any given virtual channel A→B, the *r* events occur in exactly the same order as the *s* events, because the channel preserves message order. We now require that H is informed of events that happen at any node in the order in which they happen at that node.[7] Thus, all events are matched in H's view if the number of <r, o, A, B> events equals the number of <s, o, A, B> events.

For *r* and *d* events, the number of pointers to o in receive message buffers at B is exactly the number of <r, o, A, B> events minus the number of <d, o, A, B> events. It does not matter that the order of receives may differ from the order of deletes, only that for any(o, B, E) to be false, the number of *r* and *d* events for o at B must be equal.

For + and – events, as for the *r* and *d* events, the net number of heap pointers to o at B is the number of + events minus the number of – events, and again, for any(o, B, E) to be false, the number of + events must equal the number of – events.

This optimisation depends on the fact that *d* events occur only after their corresponding *r* events, and likewise for – events after + events.

4.4.2 Reducing the number of events referred to H

We need to inform H of a + (respectively, –) event only if it causes any(o, B, E) to change from false to true (respectively, true to false). This is correct since H does not need to know the actual number of pointers at B, only whether or not there are any.

For future purposes, we observe that we can tell H independently about whether or not each *car* has any pointers to the object in question. Again, the key point is that H will correctly know whether or not there are any pointers at all.

4.4.3 Further reducing the detail required at H

Incorporating the first optimisation means that H will keep only net counts based on (o, A→B) for pointers to o sent from A to B, net counts inHeap(o, B) for pointers to o in the heap at B, and inBuffers(o, B) for pointers in message receive buffers. Can we reduce the number of counts further? One might think that the net number of copies of pointers to o, irrespective of source and destination, in transit in any channel would be adequate. However, the following counter-example, illustrated in Figure 4, shows otherwise:

Node A has one pointer to o, and no other node has any pointers to it (other than possibly H, the home node for o). The pointer is copied from node A to node X (step a in the figure) and then deleted from A (step b) and copied back from X (step c). Node A then copies the pointer to B (step d), which deletes it from its received buffer without using it (step e).

[7] This is simplified by our assumption of in-order message delivery.

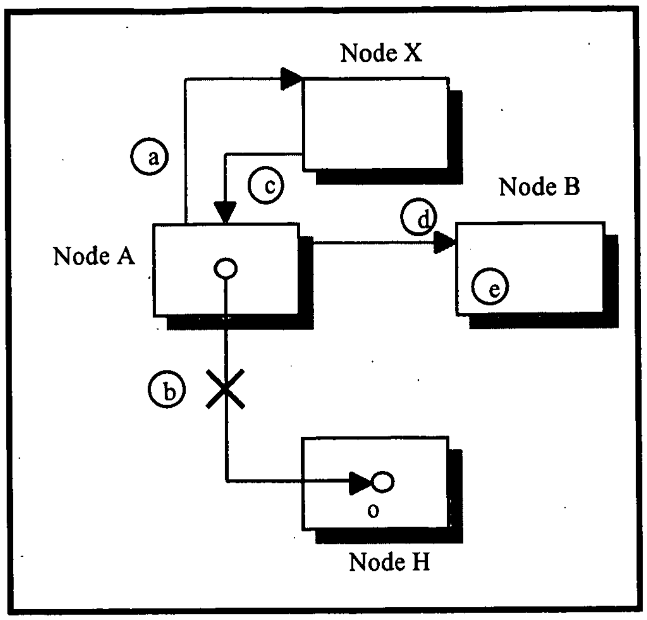


Figure 4: Optimisation Counter-example

Sequence	Node X	Node A	Node B	Step
1		<s, o, A, X>		a
2		<–, o, A>		b
3	<r, o, A, X>			a
4	<s, o, X, A>			c
5		<r, o, X, A>		c
6		<s, o, A, B>		d
7			<r, o, A, B>	d
8			<d, o, A, B>	e

Table 3: Event Sequence Corresponding to Figure 4

Table 3 shows the same sequence of events, with the dark line indicating one possible legal arrival order at node H (H has been informed of events above the line, but not of events below the line). Note that by counting only the number of sends minus the number of receives, irrespective of source and destination, H would believe falsely that there were no pointers to object o.

So, we cannot ignore both A and B in keeping (o, A→B) counts. Can we ignore B (the receiver)? No; the counter-example demonstrates that as well, since

$$\#\langle s, o, A, \ldots\rangle - \#\langle r, o, A, \ldots\rangle = 0$$

at H. Can we ignore A (the sender)? Perhaps surprisingly, the answer is *yes.*

Suppose H has net send/receive counts for a set of nodes N and absence(o, E) is true.[8] It is easy to see that if there are no pointers in transit or on other nodes, then eventually *absence* will be true. Conversely, suppose *absence* is true; we wish to show that all *s* and *r* events are paired, as well as *r* and *d* events and + and – events. In Section 4.4.1 we argued that if the inHeap(o, B) and inBuffers(o, B) counts are 0 then any(o, B, E) is false. So now we need only show that if the (o, →B) counts are also 0 then all *s* and *r* events are paired.

[8] This means that the (o, →B), the inHeap(o, B), and the inBuffers(o, B) counts for all B ε N are 0.

Firstly, observe that if any node B ε N sent a pointer elsewhere, H would know since the *s* event would precede the *d* or – event that made any(o, B, E) false. Thus a pointer cannot have been sent to a node not in N, i.e., to a node for which H did not establish a count. From the observation we further conclude that H is aware of all *s* events pertaining to o. The only way the (o, →B) counts can all be 0 is for each of these *s* events to be paired with its matching *r* event.

In sum, (o, →B) counts guarantee that H is aware of all nodes to which pointers to o have propagated, and the counts being 0 imply the required pairing of events.

4.4.4 Piggy-backing and compressing messages

As previously observed, messages informing home nodes of pointer events can be held and piggy-backed on other communications. Further, sequences of events can be compressed, especially if they are guaranteed to be delivered all at once. For example, an *r* event, then a + event, then a *d* event can be processed to compress the *r* and *d* events together. We will not pursue the issue further here, since it does not relate to correctness or completeness of our algorithms, though the performance improvements may be important in practice.

4.4.5 Combining events

It is possible to combine the *d* and – events as long as it is remembered that the *r* event acts as both a + event at the receiver and a balance for *s* in the virtual channel.[9] The effect of this is to alter the counts kept at H for pointer tracking and to reduce the *any* predicate from (#r ≠ #d) ∨ (#+ ≠ #–) to #+ ≠ #–. While this optimisation does not reduce the number of events generated dynamically, it does simplify the calculation of *any* by reducing the number of kinds of events from five to four and it reduces the size of the bookkeeping data.

If we incorporate all the optimisations above, then only two counts for each object and node are required: an *inTransit*(o, →B) count to record the number of pointers to object o sent out to node B but not yet received, and a *pointersTo*(o, B) count to record whether node B has any pointers to object o. The effect on these counts of messages arriving at H is summarised in Table 3.

Event	**Effect at H**	
	inTransit (o, →B)	**pointersTo** (o, B)
<s, o, B>	+	
<r, o, B>	–	+
<+, o, B>		+
<–, o, B>		–

Table 4: Optimised Counts at H

The inTransit count can take on any integer value including negative numbers. The pointersTo count can be only 0 or 1, and the collection of these counts for a single object o encodes what we will call the *current remembered set,* the set of nodes currently known to possess pointers to o.

5 Object Substitution Protocol

DMOS is a copying collector. When it copies an object, all references to the old copy must be updated to refer to the new copy. While DMOS does not need to make copies across nodes, we have designed the *object substitution* protocol to support cross-node

[9] Note that the *r* and + events cannot be combined since the *r* event is required to balance the *s* event.

copying (object migration). A mutator, or a collector with a different policy, may take advantage of this capability.

Any object substitution protocol must

- work while references are being updated (safety), and
- find and update all references eventually (completeness).

The above must be combined with our goal of asynchrony.

The specific goal of object substitution is to replace object o, home node H, with object o', home node H' (where H' may or may not be H), and to have all pointers to o in the entire system eventually replaced with pointers to o'.

To support object substitution, home nodes, H, maintain for each moved object o, KnownNodes(o), the set of nodes that H knows have had pointers to o since the decision was made to substitute o' for o. Likewise, all nodes maintain *object relocation tables* with entries of the form o⇒o', meaning that o has been substituted by o'.

The algorithm is described through the messages required to support it and how nodes should respond to those messages:[10]

- When o is substituted by o', H adds o⇒o' to its relocation table, and initialises KnownNodes(o) to contain those nodes X for which any(o, X, E) is true in H's current view E. Then H sends a message [m, o, o'] (m is for *move*) to each node in KnownNodes(o). Note that the *m* message should be considered an *s* event for o', but not for o, in the pointer tracking algorithm.
- When a node X receives a message [m, o, o'], it adds o⇒o' to its object relocation table. The *m* message should be treated as an *r* event of o' but not o, and the relocation table entry should count as an occurrence of o' but not of o.
- Once X has inserted a relocation table entry o⇒o', it should (at its leisure, but before deleting the table entry) replace any receive buffer or heap pointers to o with pointers to o'. Such replacement is considered a <+, o', X> event and a <–, o, X> event.
- If X receives a pointer to o while it has o⇒o' in its relocation table, it should replace o with o', causing events <r, o, X>, <+, o', X>, and <–, o, X>.
- If H has o⇒o' in its relocation table and is informed of an event of the form <s, o, X>, <r, o, X>, <+, o, X>, or <–, o, X> then it should check whether X is in KnownNodes(o); if it is not, then it should be added and an *m* message sent to X.
- Since H deletes the contents of o during the substitution, a <–, ...> event is induced at H for each pointer in o. Likewise a <+, ...> event is induced at H' for each pointer in o'. If H ≠ H' then appropriate *s* and *r* events are also induced. As soon as the contents of o have been copied either directly into o' or into a message to H' then the space occupied by o may be reclaimed. This works since the object identifier o is unique and will not be reused.
- If H ≠ H', then the creation and management of the copy requires more steps. H sends a message to H' indicating that it would like to migrate o and requesting a pointer to the copy o' that H' will allocate. This message includes the contents of o, and is considered an *s* event at H for each pointer in the contents of o, but this communication to H' should not be considered an *s* event of o to H'. H' sends back a response to H with the new pointer, which is considered an *s* event of o'. H proceeds as described above. This protocol does not allow H' to reject the request.

[10] From now on we will use the optimisations given in Section 4.

- If H ≠ H', and H receives a message to manipulate o, then in addition to the protocol steps above, H forwards the request to H'.

5.1 Cleaning up the Tables

Upon detection of absence(o, E) using the pointer tracking algorithm and completion of substitution of o' for o at H, H sends [e, o, o'] (e is for *end move*) to each node X in KnownNodes(o), deletes KnownNodes(o), and removes o⇒o' from its relocation table. This last step is a <–, o', H> event. An [e, o, o'] message is not an *s* event for either o or o'.

When node X receives [e, o, o'], it removes o⇒o' from its relocation table which is a <–, o', A> event.

Note that the substitution protocol is entirely asynchronous and never delays computation.

5.2 Correctness

The clean up condition above establishes that all pointers to o have been deleted, either by applications directly, or by having o' substituted. Thus the table clean up actions are safe. The tricky part is understanding why termination will eventually be achieved. In fact, if the number of nodes is not bounded, then pointers to o may continue to propagate ahead of the *m* messages, so termination is not guaranteed without bounding system size or somehow constraining propagation behaviour. This appears to be unavoidable.

If we bound the number of nodes to which o is propagated, it can be passed around among those nodes forever. So without the KnownNodes set and relocation tables kept at the remote nodes substitution might never terminate. That is why we introduced them. The KnownNodes set prevents sending an *m* message to the same remote node twice and also allows us to send the *e* messages to help the remote nodes clean up their relocation tables.

5.3 Multiple Substitutions

Suppose we have a series of substitutions in progress for the same object, e.g., o⇒o' and o'⇒o''. It is simplest if we maintain multiple relocation table entries and view the substitutions as happening one after the other. However, we need not notify H' of the pairs of <+, o', ...> and <–, o', ...> events, and in effect we directly substitute o'' for o. With care one could flag this directly in the relocation table.

5.4 Opaque Addressing

An obvious optimisation to the basic DMOS collector can be made if object addressing is opaque or semi-opaque, that is, if nodes maintain a mapping from external references to local addresses. Substituting an object within the node becomes a matter only of updating the local map. This is a simple operation if an indirection table is employed but may be slightly more complicated in the presence of pointer swizzling. In either case it does not entail the participation of other nodes since the external address does not change. Thus the object substitution protocol may be greatly simplified. Note also, however, that since the substituted object usually resides in a different car, appropriate – and + events usually occur for every pointer in the contents of the substituted object.

The solution does not, however, work for object migration, where we have to revert to our original method.

6 Remembered Sets, Car Collection, and Train Management

To support the DMOS collector, we refine the pointer tracking algorithm to indicate at o's home node H which *cars* have pointers to o at H. That is, H will maintain tables indicating the cars C that have one or more pointers to o, and + and – messages will indicate the cars gaining and losing any pointers to o. Another way of understanding this is that the pointer tracking algorithm is the DMOS correlate of remembered set maintenance.

In order to solve a completeness problem that we explain later, we distinguish between the current remembered set for o and the *sticky remembered set* for o. The current remembered set is as previously described in the pointer tracking algorithm, as refined to track on a per car basis. The sticky remembered set accumulates every car that is ever known by H to point to o and is deleted when o is substituted by another object. The current remembered set will thus always be a subset of the sticky remembered set.

The DMOS collector rules constrain the choices of where to move objects from a car C in order to evacuate C. DMOS uses the object substitution algorithm to accomplish those movements. C is evacuated and its space can be reclaimed once all of C's reachable objects have been copied.

DMOS does require some additional protocols, described in more detail below. Firstly, it must be able to create and delete cars and trains, and clean up any associated data structures. Secondly it must be able to detect when a *train's* sticky remembered set is empty (i.e., there are no pointers to objects in the train from outside of the train), to reclaim the entire train.

6.1 Basic Train Management

We identify each train with a pair n:A, where the positive integer n indicates the logical *birth date*[11] of the train (i.e., higher numbers are younger), and A is the node that created the train (we term it the train's *master* node). The number n is unique within the master node, thus n:A is unique within the whole system. We assume that nodes are also ordered (e.g., by some kind of node numbers), and n:A < m:B iff n < m or (n = m and A < B), i.e., lexicographic ordering. The master node A of train n:A is responsible for creating, managing, and cleaning up the train.

Although node A created train n:A, any number of nodes may contain cars of n:A. All nodes holding cars of n:A are linked together in a logical token passing ring, where each node X in the n:A ring knows its successor, written successor(X, n:A), at any given time. Initially successor(A, n:A) is A. Nodes may join or leave the train independently.

Joining a ring: If a node X wishes to create a car at X in n:A but is not currently in the n:A ring, it sends a [join, X, n:A] message to A. When A receives that message, it sends the message [succ, successor(A, n:A), n:A] to X and updates successor(A, n:A) to be X. That is, A inserts X after A in the circularly linked list of nodes in the n:A ring.

Leaving a ring: If node X has no cars of n:A but is still in the n:A ring, it can send a message [leave, X, successor(X, n:A), n:A] to its successor, to start exiting the ring. The general idea is that the [leave, ...] message propagates around the ring to X's predecessor, which then cuts X out of the ring (using the knowledge of X's successor that X thoughtfully provided in the leave message) and informs X that it has in fact been removed. In the meantime X must continue to pass messages around the ring.

However, multiple nodes may be trying to exit the ring at the same time, so the complete algorithm is a little more complicated, and we describe it according to how nodes should process leave messages. Suppose the message [leave, Y, Z, n:A] arrives at node X; X responds according to these cases and actions:

Case 1: Y = successor(X, n:A), i.e., X is Y's predecessor: X sets successor(X, n:A) to be Z (Y's successor) and sends the message [left, n:A] to Y.

Case 2: Z = X and X is not in the process of leaving the ring: X sends the message [leave, Y, Z, n:A] to successor(X, n:A).

Case 3: Z = X and X is in the process of leaving the ring: X sends the message [leave, Y, successor(X, n:A), n:A] to successor(X, n:A).

The first two cases are fairly obvious, but the third one is more subtle because it passes a modified message further along the ring. It pertains when X's predecessor, Y in this

[11] The *birth date* need not indicate a date or time but is used only to indicate relative ages of trains.

case, starts to remove itself from the ring while X is being removed. Modifying the message guarantees that Y's predecessor will cut both Y and X from the ring. The technique is general and will work for any number of simultaneous deletions from the ring. The algorithm depends on the fact that messages flowing around the ring cannot pass one another.

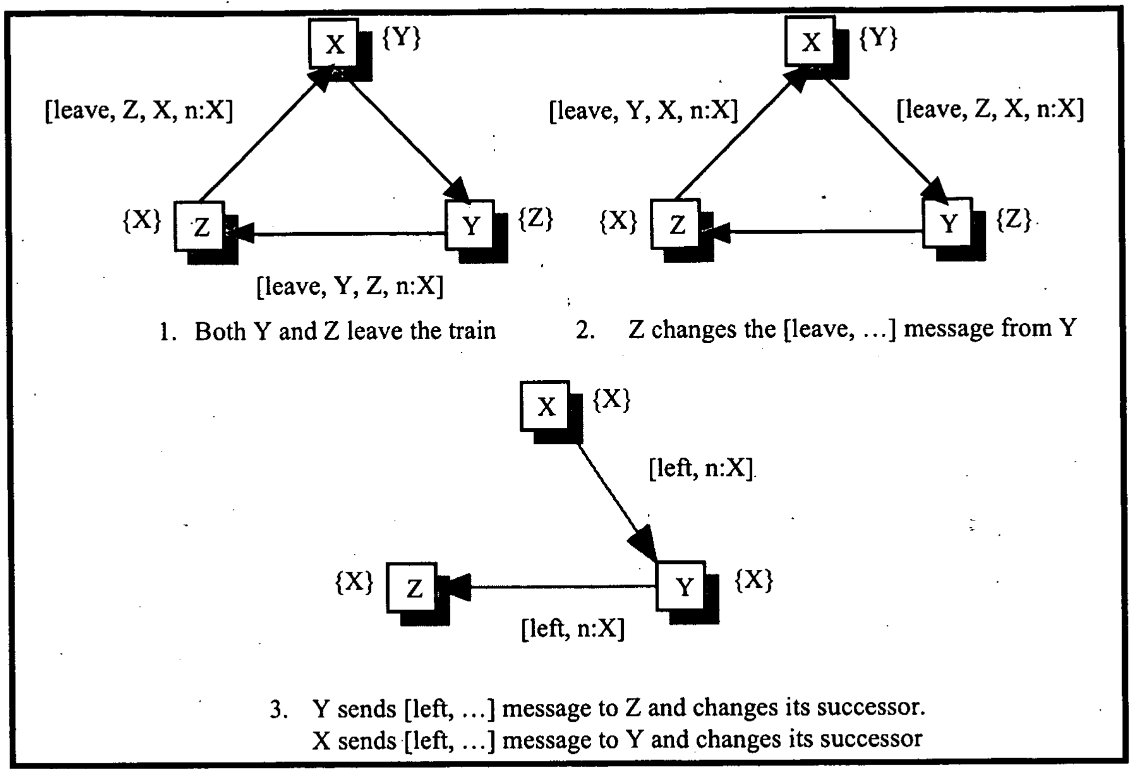

X
{Y}
[leave, Z, X, n:X]
{X}
Z
Y
{Z}
[leave, Y, Z, n:X]
1. Both Y and Z leave the train
X
{Y}
[leave, Y, X, n:X]
[leave, Z, X, n:X]
{X}
Z
Y
{Z}
2. Z changes the [leave, ...] message from Y
X
{X}
[left, n:X]
{X}
Z
Y
{X}
[left, n:X]
3. Y sends [left, ...] message to Z and changes its successor.
X sends [left, ...] message to Y and changes its successor

Figure 5: Two Nodes Leave a Train Ring at the Same Time

Figure 5 shows the sequence of [leave, ...] messages when two adjacent nodes leave a train ring at the same time. In the diagram there are three nodes X, Y, and Z, with X as the master node. The successor of a node is indicated in braces beside the node. Nodes Y and Z each send a [leave, ...] message to their respective successors. Node Z is in the process of leaving when the [leave, ...] message arrives from Y, so it alters Y's successor in the message to its own successor X and sends the amended message to X. Y knows that its successor is Z, so when Y receives Z's [leave, ...] message it changes its successor to the successor in the message and sends a [left, ...] message to Z thereby cutting Z out of the ring. Similarly the [leave, ...] message from Y arrives at X (Y's predecessor) and X sets its successor to the one in the message, cutting Y from the ring, and sends a [left, ...] message to Y.

We do need one special rule, though: A may not delete itself from the n:A ring unless and until it is the only node in n:A. This is because A is the authority for adding nodes to n:A. If A could delete itself before n:A is otherwise empty then we could end up with two independent rings.

Note that while a node is in the process of being deleted from the n:A ring, it cannot create cars in n:A. We will return to this point in the discussion of the train reclamation algorithm.

The approach we have described allows any node to create new trains without synchronising with other nodes. It also requires minimal synchronisation (with a train's master node, and only if the train is not locally represented) to add cars to existing trains. A train can even come into existence, be reclaimed, and be created again, with no ill effect.

When are trains created in DMOS? New young trains can be created at any time; we do not specify any particular policy. However, each node should have at least two trains and should work to move objects reachable from local roots to younger trains. Other than new trains, the collector may create a new representative of an existing (or previously existing) train T at a node, if local objects have references from T in their sticky remembered set. These are the only ways in which trains are created in DMOS. Note that after a certain point, the DMOS collector will not create trains of a given birth date, i.e., once that date is earlier than the birth date of any train currently in the system. In our approach, trains are cleaned up without any global knowledge. Thus, we have achieved train management that is simple, fully distributed, and minimally synchronised.

6.2 Train Reclamation

The original MOS algorithm, as well as the PMOS and DMOS algorithms, depends on being able to detect when there are no pointers into a train from outside of the train, allowing the whole train to be reclaimed at once. Such detection is trivial for MOS and PMOS because they are neither distributed nor asynchronous; as might be expected, we need a more subtle algorithm for DMOS.

The basic idea in detecting that there are no pointers into a train is to pass a *token* around the train's ring. We first describe a protocol that works if no objects can be created in the train or added to the train during detection. We later describe a problem with that restriction, and extend the basic algorithm to relax the restriction.

At any given time the token resides at a single node in the ring, and under specific circumstances is passed from the current token holder to that holder's successor in the ring. The token also bears a *value*, which indicates a node in the ring where a current round of detection of absence of external pointers began. An *external pointer* is a pointer from outside the train to an object in the train; significantly, a root pointer is considered external.

Each node X in n:A maintains a *changed bit* related to n:A, which indicates whether X has been aware of any external pointers to objects in n:A at X, since the last time the token was held by X. When X joins the ring, its changed bit is initialised to true. A's changed bit is likewise true initially. If an external pointer is entered into a sticky remembered set at X then X will set the appropriate changed bit to true.

Initial state: The token for n:A starts at node A; its initial value is A.

Starting the token: If node X holds the token, and goes from having external pointers in its sticky remembered sets to having none,[12] it sets its changed bit to false and sends the token to its successor, with value X.

Receiving the token: If node Y receives the token, it either holds it or passes it on, according to these rules:

Rule 1: If Y has external pointers in its sticky remembered sets for the train, then Y retains the token, and must wait until none of its sticky remembered sets contain external pointers, at which time Y will start the token.

Rule 2: If Y has no external pointers in its sticky remembered sets for the train, but its changed bit is true, Y passes the token, but with the value set to Y. At the same time, Y sets its changed bit to false.

Rule 3: If Y has no external pointers in its sticky remembered sets for the train, and its changed bit is false, Y passes the token with the value as Y received it. As a special case, if the received value is Y, then we have detected that there are no external pointers to the train, at any node of the train, and the train can be reclaimed.

Though we will need to extend this algorithm, let us first gain understanding of how it works under the assumption that no new objects are added to the train. We want to demonstrate that we *detect* no external pointers if and only if there *are* in fact no external pointers. Note that if at some instant of time there are no external pointers to a

[12] While no individual sticky remembered set has pointers removed, car collection causes entire sets to be deleted, thus possibly removing external pointers in sticky remembered sets at the node.

given train, none will be created in the future: since the objects are unreachable, the mutator will not create external pointers; since the collector moves objects to other trains only if they are reachable from a root or pointed to from another train, the collector will not create external pointers either.

The "if" part is easy: once there are no external pointers, the token will make at most two circuits of the ring, setting the changed bits to false on the first circuit and accomplishing its detection pass on the second circuit. Note also that since no objects are created in the train, no nodes will be joining the ring, etc.

The "only if" part of the argument is a little harder. Suppose we detect no external pointers via a token that starts and ends at node X (see Figure 6). We claim that at the time X started the token, there were in fact no external pointers (which is a stable condition). Call that time X_{n-1}; clearly X had no external pointers at time X_{n-1}. Consider any other node on the ring, Y; let Y_n be the time that the token most recently passed Y, and Y_{n-1} the time it left Y on the previous pass. Now $Y_{n-1} < X_{n-1} < Y_n$. Further Y had no external pointers at time Y_{n-1}, and, since Y's changed bit is false at time Y_n, Y had no pointers at any time between Y_{n-1} and Y_n, so Y had no external pointers at time X_{n-1}. This argument holds for all other nodes in the ring, so none of the nodes in the ring had external pointers at time X_{n-1}. Figure 6 argues this diagrammatically.[13]

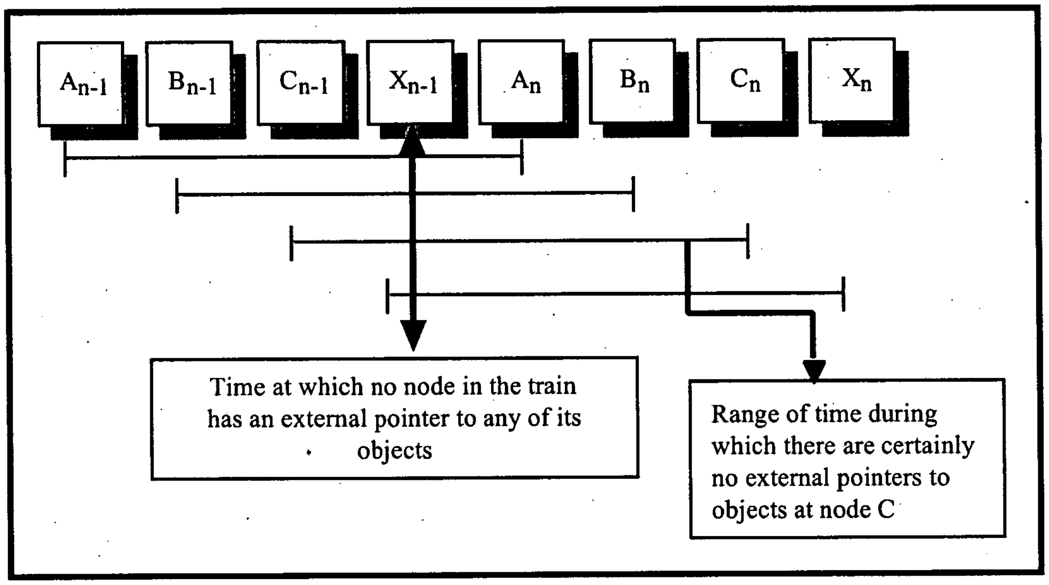

A_{n-1}
B_{n-1}
C_{n-1}
X_{n-1}
A_n
B_n
C_n
X_n
Time at which no node in the train has an external pointer to any of its objects
Range of time during which there are certainly no external pointers to objects at node C

Figure 6: Illustration of Correctness of the Train Reclamation Algorithm

We observe that our token ring algorithm is a particular kind of distributed termination algorithm, i.e., an algorithm that discovers a stable global property in a distributed system, in this case the non-existence of external pointers to objects on a set of nodes. There have been many distributed termination algorithms published [DFG83, CM86, Mattern87], and we suspect that just about any of them could be adapted and used to solve the train reclamation problem. Why, then did we choose this one? We felt that a token passing ring would be simple and convincing, even in the face of changing membership in the train. Also, we believe that the token ring's overhead is low and that its latency is not problematic in this case since train reclamation is not urgent. In any case, the particular distributed termination algorithm used is not of importance to the completeness of the DMOS collector.

13 The time argument does not rely on any notion of global time, but only on the inherent causal ordering of events in the system.

6.2.1 The Unwanted Relative Problem

The token ring algorithm just described assumes that no objects are added to the train while the algorithm is running. To prevent new objects from being added, each node always marks one or more of its oldest trains as *closed*, meaning no new objects may be created in the train, and starts or passes a train's token only if the train is closed at the node.[14] It is hard to prevent the collector from trying to move objects into a train, as shown by the following scenario (illustrated in Figure 7), which we term the *unwanted relative*:

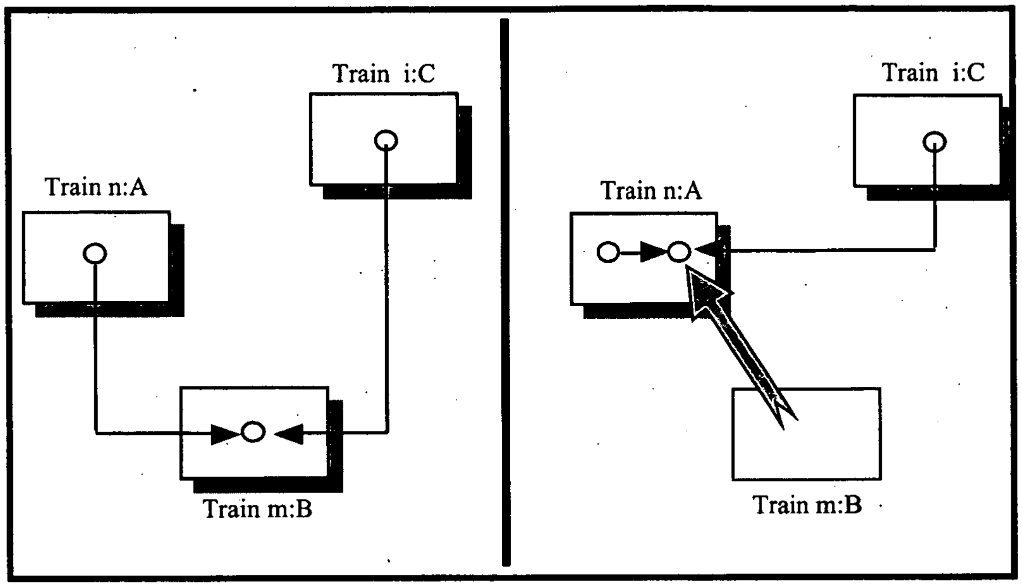


Figure 7: The "Unwanted Relative" Problem

In Figure 7, train n:A has no external pointers in its sticky remembered set but has a pointer to an object in an older train m:B. Train i:C is younger than train n:A and also has a pointer to the same object. When collecting the car in train m:B the collector moves the object into train n:A thereby creating an external pointer from i:C to n:A.

Further, note that n:A need no longer actually point to the object in m:B, since m:B's information can be out of date because of asynchrony in the system.

We considered the following design options, rejecting each for the reasons indicated:

- Disallow moving unwanted relatives into n:A. This is undesirable since it implies a synchronous inter-node protocol for the collector to check whether a relative is unwanted.
- Delay moving the relative in until the train's status is better determined. This is messy, and the delay cannot be bounded. Again, it introduces delays and dependencies into a collection process where we prefer to avoid inter-node interaction.
- Since the problem does not occur if n:A is the oldest train, attempt to reclaim a train only if it is the oldest. This would delay train reclamation needlessly; it would also require a global protocol to determine when a train is oldest.

The alternative we adopt is to introduce the notion of *epochs* of object creation in a train. Each node in the train ring associates each of its cars in the train with either the *old epoch* or the *new epoch.* When node X starts the token for n:A, it associates all of its n:A cars with the old epoch and sets its changed bit to false. Cars added at node X when the changed bit is false are added to the new epoch; cars added when the changed

[14] Receiving the token may indicate that the train is a good one to close.

bit is true are added to the old epoch. Further, if the changed bit switches from false to true at X, all n:A cars at X become associated with the old epoch. The changed bit for n:A at X is set only if X sees an external pointer to an object in an old epoch car of n:A at X.

We claim that the token ring algorithm, modified as just described, correctly detects that the old epoch n:A cars are not reachable and can be discarded. The only way in which the previous arguments could fail is if a new epoch object n points to an old epoch object o (and n is itself reachable). Since the train was closed, n was moved into the train from outside and had a pointer to o. So there was an external pointer to o at a time o was unreachable, a contradiction.

A key observation here is that at the time the token starts its last circuit, not only do all of the nodes in the n:A ring believe that there are no external pointers to their (old epoch) n:A objects, it is in fact true. That is, their perception is up to date. The reasoning is similar to that used in arguing the correctness of the pointer tracking algorithm.

6.2.2 Cleaning up Trains

Once the old epoch has been detected as unreachable, the new epoch becomes the old one, the new one becomes empty, and the system starts all over. This is accomplished by sending an [end-of-epoch, X] message around the ring, where X is the node that started the token. As each node receives the message, it deletes its old epoch cars, marks any new epoch ones as old, and passes on the end-of-epoch message. When X receives the end-of-epoch message, it restarts the token passing algorithm on the new epoch.

But this leaves the question: how do we ever get rid of a train if we can keep creating new epochs in it? One answer is that, as previously mentioned, eventually the train will become the oldest in the system and will no longer have objects moved into it. However, we believe that taking advantage of this fact requires a protocol to detect the oldest train.

The problem is somewhat easier than that since the collector can be designed so that when it moves an object from one train to another, it does so only within a single node.[15] Thus, we can decide to remove a node X (other than A) from the n:A ring if X has no objects in n:A or older trains. Once we start removing X from n:A, we disallow moving objects into n:A at X until X receives the [left, X, n:A] message, at which time it can rejoin n:A if necessary or desired. So, we prohibit some object substitutions temporarily. This temporary prohibition will not affect the collector's completeness since it merely slows the collector down.[16]

If we reach a situation where A is the only node in the n:A ring, and the n:A train is empty at A, we can delete n:A entirely. Due to asynchrony, it is possible that another node X might later request to join n:A, and we can simply recreate n:A at A at that time.

7 Collector Safety and Completeness Arguments

As with many algorithms for continuously running systems, correctness of incremental garbage collection algorithms breaks down into two distinct parts. One is safety, in this case that the collector never deletes a reachable object. The other is liveness (or progress), in this case that the collector eventually reclaims every unreachable object, which has come to be called completeness.

[15] The only constraint this imposes is that inter-node migration of objects must be done by having the new and old versions of the object in the same train, which in no way inhibits migration from node to node.

[16] Alternatively, we can thread the n:A ring through the *cars* of n:A (instead of the nodes), which supports adding new cars (at a possibly different position in the ring) while old cars are being deleted.

7.1 Safety

We now argue that the collector never discards a reachable object. Let us first consider the atomicity of object substitution. If o' is substituted for o with both o' and o at node H, H can ensure that the substitution is atomic locally. Other nodes can only pass around pointers to o, and we previously argued that pointers to o will be replaced with pointers to o', provided the number of nodes involved is bounded. Further, the substitution algorithms work by making the substitution atomically at each affected node, as the information reaches that node. Any later messages containing pointers to o are updated before the mutator can see them.

If o and o' are on different home nodes H and H', H' takes over responsibility for the migrated object as soon as the information arrives at H', and H gives up manipulating o as soon as it sends it to H'. There is a period of time during which H does not know the new identifier o' for o at H', and will have to refer application requests concerning o to H' under the identifier o, but H' will use its relocation table to rewrite the incoming pointer to o', so everything works out without indefinite waits.

Our point is that object deletion related to object substitution is not a problem. Observe also that since object relocation tables are considered to contain pointers to the new objects (o' in the example), the new object will not be deleted until we have cleaned up all pointers to the old one, or the new object is itself substituted.

Where else does the algorithm delete objects? In car collection and train reclamation. The car collection algorithm discards only objects not reachable from outside the car. These objects must be in the absence(o, E) condition as used in the pointer tracking algorithm. Therefore, they can be reached only through other objects on the same node; but remembered sets are always accurate with respect to references from our own node, so there is no path to the objects deleted.

The train reclamation algorithm was separately argued: the old epoch objects in the train were unreachable from outside the train, and also unreachable from new epoch objects in the train, and so are unreachable and it is correct to discard them.

7.2 Completeness

The completeness argument is similar to those found in Bishop [Bishop77], Hudson and Moss [HM92], Moss et al. [MMH96], and Seligmann and Grarup [SG95]. The argument proceeds in two main steps. Firstly, we show that the oldest train will eventually be evacuated and secondly that all garbage in trains present at a given time t will be eventually collected.

First we argue that the oldest train will be eventually collected. Consider the set of cars C in a train T at time t, and consider the situation after each car in C has been collected. If there are no external sticky remembered set entries with pointers into train T then the entire train is eventually collected by the train reclamation protocol. If there are such entries then as we collect the cars these objects are evacuated thus showing progress collecting the train during each pass through the cars of the train. If the train is the oldest then no new objects can be created in or moved into the train so each pass through the cars reduces the number of objects in T and by induction T will eventually be completely evacuated. Note that the stickiness of sticky remembered set entries (i.e., that the sticky remembered set may be a superset of the current remembered set) is crucial to guaranteeing progress in the case that there are external sticky remembered set entries, since it is one of these entries that will be used to identify an object to be moved out of the train. Otherwise the mutator could move pointers around such that there were no current external pointers at any car when we collected that car, but that there were such pointers for other cars. This is the problem identified by Seligmann and Grarup [SG95].

We now argue that garbage is eventually reclaimed. Consider a time t; let G be the set of unreachable objects at t, and S be the set of trains existing at that time. Remembered set entries at time t can only be from trains in S. Since garbage is immutable, remembered set entries for objects in G will *never* mention trains not in S so garbage will not move to trains not in S. Eventually the oldest train in S will be evacuated, and then the next oldest and thus eventually every train in S will be evacuated, and at that point all objects in G will have been collected. The final inductive step in the argument depends on two additional properties: that mutators do not allocate new objects in the oldest train, and that only a finite number of trains can be created of ages intermediate between two trains (which is ensured because train numbers n:A are formed from positive integers n, and node names A from a finite set of nodes).

8 Related Work

Work in distributed garbage collection has become increasingly active as distributed systems become increasingly important; we do not attempt to cover all related work, but focus on the most relevant contributions. Plainfossé and Shapiro [PS95] offer a survey. Previous and ongoing research in this area falls into three categories: object migration, reference counting, and tracing. Some proposed algorithms are hybrids that combine these techniques. We will discuss the approaches one at a time and indicate how they have been combined.

8.1 Migration

Bishop presents a non-distributed garbage collection algorithm that divides the heap into multiple areas [Bishop77]. Users specify the area in which each object is allocated. These areas are designed to be garbage collected individually so that the collections do not interfere with processes that do not use the area being collected. In order to allow independent collection, each area keeps track of pointers both into the area and out of the area. Referencing an object in another area is accomplished using a level of indirection.

Bishop points out that related areas could be collected at the same time. He handles multiple area cycles of garbage either by collecting all areas involved in the cycle at the same time, or by moving objects to consolidate the cycle of objects into one area. He presents an inductive proof to show that his technique of moving objects guarantees that all unreachable objects are collected. Bishop does not bound the size of an area or provide ways to collect individual areas incrementally. The obvious distributed version of Bishop's algorithm uses one area per node, which requires object migration to collect inter-node cyclic garbage. Our algorithm does not require migration and is also incremental.

8.2 Reference Counting

Reference counting has been used to collect distributed objects. The advantage of reference counting is that the rules appear simple; but reference counting alone cannot guarantee completeness (because of cyclic data structures), and making a copy of a reference requires contacting the owner of the referent object.

Bevan [Bevan87] and Watson and Watson [WW87] introduce a refinement to traditional reference counting called *weighted reference counting* where each reference count is divided into a partial weight and a total weight. Unlike the DMOS collector, weighted reference counting avoids the need to send a message to the owner of an object whenever an object reference is passed from node to node. However, it is still a reference counting scheme and suffers from inability to collect cycles.

In *reference listing* [BEN+93] an entry is maintained for each node holding a reference to an object, while reference counting maintains only the count of such references. Reference listing uses more space than reference counting, but messages are idempotent so the system is resilient against message duplication and loss. Again reference listing does not handle cyclic garbage.

Both reference listing and reference counting schemes require that cyclic garbage be rare and sufficient memory be provided to tolerate the leakage. Extensions to reference listing to handle cycles include *optimised weighted reference counting* augmented with background global tracing [Dickman91], and reference listing with partial tracing [RJ96].

8.3 Tracing

Hughes [Hughes85] uses time stamps based on global time to trace live objects. Each trace initiated on a node uses the time stamp to mark objects. Each outgoing pointer uses the time stamp whenever it propagates the trace to other nodes. The algorithm requires a globally synchronised clock, and message delivery time must be bounded. Given these requirements, Hughes shows that any object with a time stamp older than a certain time is garbage and can be collected. The termination algorithm used by Hughes is not scalable and reclamation of distributed garbage can be blocked until the slowest node in the system performs a local garbage collection.

Liskov and Ladin [LL86] propose using a centralised server to calculate global accessibility of objects. The idea is that each node informs the centralised (but possibly replicated) server of any pointers into and out of the node. The local collector is responsible for determining the connectivity between the incoming and outgoing references. Rudalics [Rudalics90] points out an error in the original algorithm that is corrected by Ladin and Liskov [LL92] using an adaptation of Hughes's time stamp algorithm. Their solution also uses the centralised server clock to simplify Hughes's termination algorithm.

Lang, Queninnec, and Piquer [LQP92] propose a technique where spaces (or nodes) are grouped. Any garbage cycle completely within a group is collected using a mark/sweep algorithm. The groups can be hierarchically ordered so that increasingly large groups are traced. Ultimately, the entire system needs to be traced in order to collect garbage not located entirely within a previously associated group. This is therefore not scalable and requires a considerable amount of co-ordination between the nodes. Maheshwari and Liskov [ML97] claim that the algorithm will not terminate correctly if the object graph is mutated concurrently with tracing.

Ferreira and Shapiro [FS96] propose a system that allows replication of segments at multiple sites. Each segment maintains a list of incoming and outgoing pointers and is traced using these pointers as roots. Segments that appear at the same site are collected together so cyclic structures that span segments can be collected only if they are gathered at a single site. The co-ordination of segments is not a problem since replication is assumed.

Maheshwari and Liskov [ML97] describe a partitioned garbage collector that piggy-backs global marking with the marking of partitioned data. Their scheme is guaranteed to terminate correctly, and while not as yet distributed, is optimised for efficient tracking of a partition's incoming and outgoing pointers.

8.4 Garbage Tracing

Vestal [Vestal87] suggests selecting objects suspected of being part of a cyclic garbage structure, and on a trial basis decrementing the reference count. If this causes all connected objects' counts to drop to zero, then the structure is garbage.

Lins and Jones [LJ91] propose combining weighted reference counting with mark and sweep where the marking is not started with roots but with any object that experiences a reference deletion. The reference count is copied and then decremented. If the trace returns to the start then the object is part of a cyclic graph and can be deleted. While this does collect cyclic garbage it appears to be expensive.

Maeda et al. [MKI+95] and Fuchs [Fuchs95] suggest techniques where potentially cyclic garbage is traced to see if it reaches a root. Fuchs traces the inverse graph to see if it reaches a root while Maeda et al. trace potential garbage to see if it forms an isolated cycle.

All these schemes attempt to discover garbage and suffer from the same difficulty: they need a heuristic to select suspected cyclic garbage. There are no completeness arguments for any of these schemes and all could result in much tracing of live objects.

9 Conclusions

We have presented a new garbage collection algorithm for distributed systems, DMOS (Distributed Mature Object Space). It is unique among distributed collectors in that it is safe, complete, non-disruptive, incremental, scalable, and non-blocking, as defined in the introduction. DMOS is an advance in that no prior distributed collector has possessed all these desirable properties. DMOS thus overcomes significant limitations in previous collectors: it is complete (unlike reference counting and partial tracing techniques), it is non-disruptive, incremental, and scalable (unlike global tracing), and it does not require object migration.

Like the MOS and PMOS algorithms on which DMOS is based, each collection processes a bounded size region of objects, in this case on a single node, copying them to other regions, according to a set of rules that ultimately guarantee that all garbage is collected. We track cross-region and cross-node pointers, using a distributed termination algorithm to detect when an object has no more references. We also introduce a distributed termination algorithm to detect when a distributed set of regions (a train) has no pointers into it from outside, and distributed algorithms for managing trains.

Interesting work remaining to be done including implementation and practical evaluation, algorithmic performance analysis, and extensions to tolerate node and communications failures, which we intend to address in future work.

10 Acknowledgements

We thank Peter Bailey, Huw Evans, Al Dearle, and anonymous OOPSLA referees for their constructive comments regarding this paper. The work was supported by NSF grants IRI-9632284 and INT-9600216 and by EPSRC Grant GR/J67611.

11 References

[Bakker+87] W. Jacobus Bakker, L. Nijman, and Philip C. Treleaven, editors. *Parallel Architectures and Languages Europe*, numbers 258, 259 in *Lecture Notes in Computer Science*, Springer-Verlag, June 1987.

[BC92] Bekkers and Cohen, editors. In *Proceedings of the International Workshop on Memory Management*, St. Malo, France, 1992. Published as number 637, *Lecture Notes in Computer Science*, Springer-Verlag, 1992.

[BEN+93] Andrew Birrell, David Evers, Greg Nelson, Susan Owicki, and Edward Wobber. Distributed Garbage Collection for Network Objects Digital Equipment Corporation, Systems Research Center Tech. Rep. 116, 15 December 1993.

[Bevan87] David I. Bevan. Distributed garbage collection using reference counting. In [Bakker+87].

[Bishop77] Peter B. Bishop. Computer systems with a very large address space and garbage collection. Ph.D. thesis, published as Technical Report MIT/LCS/TR-178, Massachusetts Institute of Technology, 1977.

[CM86] K. M Chandy and J. Misra. An example of stepwise refinement of distributed programs: quiescence detection. ACM Trans. on Prog. Lang. and Systems 8,326 (1986).

[CWZ94] Johnathan E. Cook, Alexander L. Wolf, and Benjamin G. Zorn. Partition selection policies in object database garbage collection. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD '94)* (Minneapolis, MN, May 1994), pp. 371-382.

[DFG83] E. W. Dijkstra, W. H. J. Feijen and A. J. M. van Gasteren. Derivation of a termination detection algorithm for distributed computation. Information Processing Letters 16, 217 (1983).

[Dickman91] Peter Dickman. Distributed object management in a non-small graph of autonomous networks with few failures. PhD thesis, University of Cambridge, United Kingdom, September 1991.

[Fidge96] C. J. Fidge. Fundamentals of distributed system observation. IEEE Software, 13(6):77-83, November 1996.

[FS96] Paulo Ferreira and Marc Shapiro. Larchant: Persistence by reachability in distributed shared memory through garbage collection. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, IEEE Press, 1996.

[Fuchs95] Matthew Fuchs. Garbage collection on an open network. In [IWMM95], pp. 251-266.

[HM92] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In [BC92].

[Hughes85] R. John M. Hughes. A distributed garbage collection algorithm. In *Proceedings of the 1985 Conference on Functional Programming and Computer Architecture*, number 201, *Lecture Notes in Computer Science*, pp. 256-272, Springer-Verlag, 1985.

[IWMM95] *Proceedings of the 1995 International Workshop on Memory Management* (Kinross, Scotland, United Kingdom). Published as number 986, *Lecture Notes in Computer Science*, Springer-Verlag.

[LJ91] Rafael D. Lins and Richard E. Jones. Cyclic weighted reference counting. Technical Report 95, University of Kent, Canterbury, United Kingdom, December 1991.

[LL86] Barbara Liskov and Rivka Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Fifth ACM Symposium on the Principles of Distributed Computing*, pp. 29-39, 1986.

[LL92] Rivka Ladin and Barbara Liskov. Garbage collection of a distributed heap. In *Proceedings of the International Conference on Distributed Computing Systems*, IEEE Press, 1992.

[LQP92] Bernard Lang, Christian Queinniec, and Jose Piquer. Garbage collecting the world. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 39-50, ACM Press, 1992.

[MKI+95] M. Maeda, H. Konake, Y. Ishikawa, T. Tomokiyo, A. Hori, and J. Nolte. On the fly global garbage collection based on partly mark-sweep. In [IWMM95], pp. 283-296.

[Mattern87] F. Mattern. Algorithms for distributed termination detection. Distributed Computing, 2,161 (1987).

[ML97] Umesh Maheshwari and Barbara Liskov. Partitioned garbage collection of a large object store. In Proceedings of ACM SIGMOD '97, Phoenix, Arizona, 1997.

[MMH96] J. Eliot B. Moss, David S. Munro, and Richard L. Hudson. PMOS: A complete and coarse-grained incremental garbage collector for persistent object stores. In *Proceedings of the 7th International Workshop on Persistent Object Systems*, pp. 140-150, Morgan Kaufmann, 1996.

[PS95] David Plainfossé and Marc Shapiro, A Survey of Distributed Garbage Collection Techniques. In *Proceedings of International Workshop on Memory Management*, Kinross, Scotland, pp. 211-249, September 1995.

[RJ96] Helena Rodrigues and Richard Jones. A Cycle Distributed Garbage Collector for Network Objects. In *Proceedings of 10th International Workshop on Distributed Algorithms* (WDAG '96) Bologna (Italy) 9-11 October 1996.

[Rudalics90] Martin Rudalics. Correctness of distributed garbage collection algorithms. Technical Report 90-40.0, Johannes Kepler Universitat, Linz Austria, 1990.

[SG95] Jacob Seligmann and Steffen Grarup. Incremental mature garbage collection using the train algorithm. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '95)* (Aarhus, Denmark, August 1995), no. 952 in *Lecture Notes in Computer Science*, Springer-Verlag, pp. 235-252.

[Vestal87] S. C. Vestal. Garbage collection: An exercise in distributed, fault-tolerant programming. PhD thesis, University of Washington, Seattle, Washington, January 1987.

[WW87] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architecture. In [Bakker+87], pp. 432-443.

[Wilson92] Paul R. Wilson. Uniprocessor garbage collection techniques. In [BC92].